Allgemein

- Physiologischer Hintergrund:
  - Zäpfchen: Licht (ca. 100 Mio.)
  - Stäbchen: Farbe (rot, grün, blau) (ca. 6 Mio.)

Farbmodelle

- RGB:
  - Rot, Grün und blau
  - Additives & multiplikatives Verfahren
  - wird verwendet in Monitoren
  - 8-bit unsigned integer
- CMY
  - Cyan, Magenta, Yellow
  - Subtraktives Verfahren -> Farben werden zu weißem Hintergrund hinzugefügt
  - verwendet in Druckern
  - Erweiterung CMYK:
    - Graustufe wird hinzugefügt
    - K = min(C, M, Y) → C‘ = C – K, ...
- HSV
  - Hue (Farbton), Saturation (Sättigung), Value (Brightness)
  - basiert auf RGB

Gamma

- CRT-Displays:
  - lineares Wachstum bei Spannung != lineares Wachstum der Helligkeit
- zur Korrektur eingeführte Gamma-Berechnung
  - $\gamma \sim 2{,}2$ → bei Bildschirm kommt x^{γ} an
  - zur Korrektur: $x_{out} = x_{in}^{1/\gamma}$
- in modernen Displays immer noch Simulation von Gamma-Korrektur

Triangles and Pixels

- Dreiecke sind primitive Flächen → alle möglichen Polygone können zu Dreiecken konvertiert werden
- Verfahren zum Speichern:
  - sequenzielle Liste:
    - alle Vertizes von jedem Dreieck in einer Liste speichern
    - `N * 3 * sizeof(vertex)` bytes
  - Indexed Face Set (IFS):
    - jeder Punkt wird einfach gespeichert, für Dreiecke nur Indizes der Vertizes
    - `N * 3 * sizeof(unit) + Nvertices * sizeof(vertex)` bytes

Rasterisierung

- Single-Sample-Rasterisierung:
  - Mittelpunkt des Pixels muss in Dreieck liegen
    - Strecken berechnen & normieren $d_{ab} = \frac{b-a}{\|b-a\|}$
    - Normalen von Strecken berechnen & ggf. normieren $n(d) = \begin{pmatrix} -d_y \\ d_x \end{pmatrix}$
    - Punkt in Relation zu Strecke setzen $p - a$
    - Skalaprodukt ausrechnen $\langle n(d) | p - a \rangle$
      - positiv: Punkt in Dreieck
      - negativ: Punkt nicht im Dreieck

- Signed-Distance Rasterisierung *Convexe* Polygone
  - Bounding Box um Dreieck → nur Pixel prüfen, die in Frage kommen können
  - Winding Order: CCW (Counter Clock Wise)
  - Hierarchisch: Divide & Conquer → Unterteilung der Bounding Box in immer kleinere Vierecke
    - wenn Eckpunkte alle innerhalb: accept
    - wenn keine Kante Dreieck schneidet: reject
- Scanline *Concarve* Polygone
  - für jeden Höhenwert wird ein Pixel auf jeder Linie gewählt
  - danach wird links nach rechts zwischen gewählten Pixeln aufgefüllt
    - EdgeTable (ET) aufstellen → Sortierung nach Y_{lower}
      - Inhalt: Y_{lower}, X_{lower}, Y_{upper}, m
    - AET: immer 2 Kanten raussuchen, bis Y_{upper} erreicht wird, immer eine Stufe höher gehen & Punkte markieren / rausschreiben

Interpolation

- Lineare Interpolation bzw. lerp:
  - $p_z = (1-t) * a_z + t * b_z = a_z + t * (b_z - a_z)$
  - $t_{AC}(P) = \frac{P_y - A_y}{C_y - A_y}$ → P liegt in diesem Fall auf $\overrightarrow{AC}$
- Baryzentrische Koordinaten:
  - Interpoliert von den verschiedenen Eckpunkten, z.B. für Farbzuweißungen
  - $\mathrm{u} = \mathrm{b} - \mathrm{a};\ \mathrm{v} = \mathrm{c} - \mathrm{a};\ p = a + \begin{pmatrix} u_x & v_x \\ u_y & v_y \end{pmatrix} * \begin{pmatrix} \beta \\ \gamma \end{pmatrix}$
  - $\mathrm{p} = \mathrm{a} + \beta * (\mathrm{b} - \mathrm{a}) + \gamma * (\mathrm{c} - \mathrm{a})$

Transformationen

- Affine:
  - Lines werden zu Lines, Parallelität bleibt bestehen, Größenverhältnisse bleiben gleich
  - dazu zählen Verschiebung, Skalierung, Scherung und Drehung

$$\mathbf{A} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}, \mathbf{t} = \begin{pmatrix} t_x \\ t_y \\ t_z \end{pmatrix}.$$ Verschiebung

$$\mathbf{A} = \begin{pmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & s_z \end{pmatrix}, \mathbf{t} = \mathbf{0}.$$ Skalierung

$$\mathbf{A} = \begin{pmatrix} 1 & 0 & d_x \\ 0 & 1 & d_y \\ 0 & 0 & 1 \end{pmatrix}, \mathbf{t} = \mathbf{0}.$$ Scherung

$$\mathbf{A} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\phi & -\sin\phi \\ 0 & \sin\phi & \cos\phi \end{pmatrix}, \mathbf{t} = \mathbf{0}.$$ X-Rotation

$$\mathbf{A} = \begin{pmatrix} \cos\phi & 0 & \sin\phi \\ 0 & 1 & 0 \\ -\sin\phi & 0 & \cos\phi \end{pmatrix}, \mathbf{t} = \mathbf{0}.$$ Y-Rotation

$$\mathbf{A} = \begin{pmatrix} \cos\phi & -\sin\phi & 0 \\ \sin\phi & \cos\phi & 0 \\ 0 & 0 & 1 \end{pmatrix}, \mathbf{t} = \mathbf{0}.$$ Z-Rotation

- Transformationen können kombiniert werden $T_2(T_1(x)) = A_2(A_1 * x + t_1) + t_2$
- Homogene Koordinaten:
  - Translationsmatrix wird um eine Koordinate erweitert:
    - homogenisieren: $\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} x + \begin{pmatrix} t_1 \\ t_2 \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} & t_1 \\ A_{21} & A_{22} & t_2 \\ 0 & 0 & 1 \end{pmatrix} x = \begin{pmatrix} x \\ y \\ w \end{pmatrix}$
    - dehomogenisieren: $H \begin{pmatrix} x \\ y \\ w \end{pmatrix} = \begin{pmatrix} \frac{x}{w} \\ \frac{y}{w} \end{pmatrix}$
  - Kombination durch einfache Multiplikation der Matrizen (von hinten)
    - z.B. $C = T * S$ → Skalierung **vor** Translation, deshalb T * S und nicht S * T
- Normalen müssen renormalisiert werden
  - zum renormalisieren werden Normalen mit der Inversen der Transformationsmatrix transformiert
  - $n' = (A^{-1})^T * n$

- Rotationen:
  - immer gleiches Schema:
    - Drehachse bleibt gleich
    - bei den anderen: $\begin{matrix} cos & -sin \\ sin & cos \end{matrix}$ → bei y als Achse wird – vertauscht
  - Euler-Winkel:
    - **Jede Rotation kann durch 3 Rotation um die Hauptachsen (X, Y, Z) erreicht werden**
    - willkürlich kombinierte Rotation um die 3 Achsen, z.B. um Z, Y und Z
    - Vorteile: Einfache Implementierung, Einfach zu benutzen, nur 3 Winkel nötig
    - Nachteile: Animationen nicht einfach, Gimbal Lock
    - Gimbal Lock:
      - Bei Rotation um 90° wird eine Achse unbrauchbar
      - anschließend sind 2 Ursprungsachsen übereinander
  - Quaternionen:
    - 2D: $\cos\varphi + i * \sin\varphi$
    - Quaternion Punkt: $\hat{p} = (p_v, 1)$
    - Quaternion Winkel: $\hat{q} = (u_q * \sin\frac{\varphi}{2}, \cos\frac{\varphi}{2})$ mit $\mathbf{u_q}$ als Rotationsachse
    - Conjugate Quaternion: $\hat{q}^* = (-q_v, q_w)$
    - Multiplikation: $\hat{p}\hat{q} = (p_v \times q_v + p_v q_w + p_w q_v,\ p_w q_w - \langle p_v | q_v \rangle)$

Koordinatensysteme

- „World" Koordinatensystem (globales) wird definiert, anschließend alle Objekte anhand von Transformationen (Plazierung, Orientierung, Skalierung) in der Welt positioniert
  - Lokales auf globales Koordinatensystem setzen, dann rotieren, skalieren und an geeignete Position schieben
    - Multiplikation des Punkts mit jeweiligen Matrizen
- alle Transformationen, die gebraucht werden, um lokales System auf globales zu übertragen, werden in Matrix M zusammengefasst
  - durch Multiplikation von Punkt p mit M kann **lokaler Punkt in globales** System übertragen werden
  - durch Multiplikation von Punkt q mit M^{-1} kann **globaler Punkt in lokales** System übertragen werden
  - zur Invertierung einer starren Transformation: $(M_t M_r)^{-1} * q = M_r^T M_{(-t)} * q$
    - transponierte Rotationsmatrix * invertierter Translationsteil * Punkt im globalen System

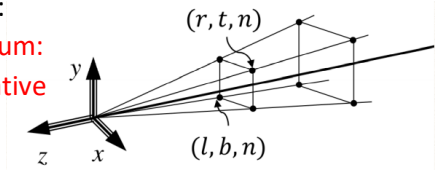
Viewing

- Virtueller Beobachter = Kamera hat neues Koordinatensystem
- Objekte werden von World in Eyespace übertragen → Projektion der Punkte auf 2D-Fläche
  - Right-handed Koordinatensystem
- **Eye Space**
  - Matrix E ist Kombination aus Translation T und Rotation R: $E = T * R$
    - Punkt wird zuerst um Ursprung gedreht, dann verschoben
  - Viewing Direction g und Up-Vector t für Rotation
    - $w = \frac{-g}{\|g\|}, u = \frac{t \times w}{\|t \times w\|}, v = w \times u$ $\quad R = \begin{pmatrix} u_x & v_x & w_x & 0 \\ u_y & v_y & w_y & 0 \\ u_z & v_z & w_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, T = \begin{pmatrix} 1 & 0 & 0 & o_x \\ 0 & 1 & 0 & o_y \\ 0 & 0 & 1 & o_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$
- **normalized device coordinates (NDC)**
  - Maximale Abmessungen werden auf Koordinaten angewandt und auf [-1, 1] Intervall gemapped (für HW)
  - Tiefe wird mit einberechnet, dafür gibt es Abmessungen n und f
  - Projektionsmatrix berechnet NDC-Koordinaten für Punkt
- **Viewport Coordinates**
  - Viewport-Größe wird in w und h angegeben, NDC-Koordinaten werden damit verrechnet & so an Viewport angepasst
  - Matrix W zur Berechnung der viewport-Koordinaten

- **orthographische Projektion**
  - keine Skalierung der Objekte
  - Projektionsmatrix hat „leeren“ homogenen Teil, also nur Einberechnung der Translation
- **perspektivische Projektion**
  - Skalierung auf n abhängig von Abstand zur Kamera
  - Foreshortening: Dinge werden in der Distanz kleiner
  - Projektionsmatrix 2D:
    - homogener Anteil nicht mehr 0 0 1, sondern Einberechnung y-Wert für Entfernung
    - Translationsteil: y-Koordinate wird mit -1 einberechnet, damit Entfernung nicht verloren geht

$$H(\mathbf{P}\,\mathbf{v}) = H\left(\begin{pmatrix} a_{00} & a_{10} & t_x \\ a_{01} & a_{11} & t_y \\ w_x & w_y & w_t \end{pmatrix}\begin{pmatrix} v_x \\ v_y \\ 1 \end{pmatrix}\right) \qquad \mathbf{P} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & -1 \\ 0 & 1 & 0 \end{pmatrix}$$

  - Projektionsmatrix 3D:
    - The Viewing Frustum:
      - z geht ins Negative



    - Einberechnung von z-Anteil, Berechnung durch Abmaße Höhe h und Weite w
      - near distance n, aspect ratio a, field of view θ
        - $w = a * n * \tan\frac{\theta}{2} \qquad = r = l$
        - $h = n * \tan\frac{\theta}{2} = b = t$
        - $a = \frac{w}{h}$
    - Berechnung von z‘ zum Bestimmen, welche Objekte gezeichnet werden sollen
      - z‘ bezeichnet Koordinate z im NDC
      - $z' = -\frac{n+f}{n-f} - \frac{2*f*n}{z*(n-f)}$
      - wenn z‘ < -1 oder z‘ > 1, dann ist das Objekt außerhalb der Sicht
      - near plane liegt bei z‘ = -1, far plane liegt bei z‘ = 1

Clipping

- nur sichtbare Triangles sollen gerendert werden
- muss im Eye Space mit Grenzen gemacht werden, da sich sonst die Geometrie verändert
- in NDC wäre Clipping einfacher, aber Geometrie verändert sich, wenn z > 0 ist
- Ordnung von Triangles:
  - Depth Buffer:
    - speichert den NDC-Z-Wert für jeden Pixel
    - wenn neuer Z-Wert niedriger ist als gespeicherter Z-Wert wird Pixel überschrieben, sonst nicht

$$\{\triangle_{local}\} \xrightarrow{\boldsymbol{M}} \{\triangle_{world}\} \xrightarrow{\boldsymbol{V}} \{\triangle_{eye}\} \xrightarrow{\boldsymbol{P}} \{\triangle_{clip}\} \xrightarrow{dehomo-genisieren} \{\triangle_{NDC}\} \xrightarrow{W} \{\triangle_{viewport}\} \xrightarrow{rasterization} [\![\square]\!]$$

OpenGL

- Idee: IFS auf GPU Speichern, Rendering für komplettes Modell auf einmal starten
- Vertex und Index Buffers werden angesprochen
  - Buffer auf RAM wird erstellt durch `glGenBuffer(size, *buffer)`
  - Binden des Buffers an GPU Buffer `glBindBuffer(target, buffer)`
    - Bindung muss am Ende wieder aufgelöst werden
    - target: für Vertices GL_ARRAY_BUFFER, für Indizes GL_ELEMENT_ARRAY_BUFFER
  - Daten von Buffer auf GPU laden `glBufferData(target, size, *data, flags)`
- **Alternativ**: Benutzen von Vertex Array Objects, die alle bindings und attribute links beinhalten
  - Erstellen durch `glGenVertexArrays(size, *vaos)`
  - Binden des Vertex Array Objects `glBindVertexArray(array)`
- Vorbereitung zum Rendern
  - Vertex Stream aktivieren `glEnableVertexAttribArray(index)`
    - Vertex Stream: Buffer wird an GPU Stream Input angebunden
  - Vertex Stream spezifizieren `glVertexAttribPointer (index,size,type,normalized,stride,*data)`
    - GL_ARRAY_BUFFER gibt die jeweiligen Vertices an den Shader
- Starten vom Rendering
  - Rendern mithilfe von Vertices `glDrawArrays(mode, first, count)`
  - Rendern mithilfe von Indizes & Vertices `glDrawElements(mode, count, type, *indices)`
    - `*indices` wird auf 0 gesetzt, dadurch wird GL_ELEMENT_ARRAY_BUFFER benutzt
  - Typen für beide Methoden: GL_TRIANGLES, GL_QUADS...

Shaders

- Vertex Shader
  - läuft für jeden einzelnen Vertex
  - transformiert Punkt durch Renderpipeline bis zum Clip Space
- Fragment Shader
  - zuständig für die Einfärbung von Pixeln
  - Signed Distance Verfahren läuft im Fragment Shader
  - außerdem Berechnung der Farbe
- Shader werden in eigenen Files geschrieben, aber müssen in OpenGL erstellt werden
  - Erstellung `glCreateProgram()`
  - Shader Stage zuweißen `glCreateShader(shader_type)`
    - `shader_type` kann `GL_VERTEX_SHADER, GL_FRAGMENT_SHADER`... sein
  - Kompilierung des Shaders: Zuweißung `glShaderSource(shader,count,**string,*length)`
  - Kompilierung `glCompileShader(shader)`
  - Error-Check `glGetShaderiv(shader, param_name, *values)`
  - Shader zu Programm zuweißen `glAttachShader(program, shader)`
  - Programm danach linken `glLinkProgram(program)`
  - Resultat checken `glGetProgramiv(program, param_name, *values)`
  - Programm benutzen `glUseProgram(program)`

OpenGL Shading Language (GLSL)

- die `main()`-Funktion wird jedes Mal, wenn der Shader aufgerufen wird, ausgeführt
- Versionsspezifizierung `#version 130`
- Spezifiziert über Main-Funktion
  - Input `in vec4 local_vertex`
  - Output `out vec4 out_color`
  - Daten, die unabhängig vom Punkt sind `uniform mat4 proj`
- main-Funktion `void main () {...}`
- implizite output-Variable für Punkt `gl_Position`

Lighting

- Physikalische Licht – Flächen Interaktionen:
  - Absorption Licht wird absorbiert
  - Transmission Licht wird weitergeleitet
  - Reflektion Licht wird reflektiert
  - Refraction Licht wird gebrochen
  - Scattering ungleichmäßige Reflektion
- lokale Beleuchtung
  - Licht wird nur von einem Punkt aus betrachtet
  - der Rest der Szene ist irrelevant
  - kein indirektes Licht
- globale Beleuchtung
  - Lichtaustausch zwischen Objekten & Lichtquellen
  - Schatten (soft Shadows)
  - Licht kann wandern & von Objekten reflektiert werden → indirektes Licht
- **Phong Lighting**
  - nutzt zum Berechnen der Beleuchtung die Richtung der Lichtquelle, die Richtung der Kamera und die Normale des aktuellen Fragments
    - die Kamera- und Lichtrichtung sind im Voraus bekannt
  - Lichttypen:
    - Directional light source unendlich weit entfernte und große Lichtquelle
      - parallelle Lichtstrahlen
      - Lichtrichtung l ist konstant
    - Point light source Lichtquelle ist Punkt im Raum
      - Licht wird von Punkt p aus in alle Richtungen gestrahlt
      - Lichtrichtung l von Punkt x aus: $l = \frac{p-x}{\|p-x\|}$
      - Die Distanz zum Licht bestimmt die Intensität
    - Spot light source
      - Variante des Punktlichts
      - Kegel um Licht → Öffnungswinkel und Richtung sind Parameter
      - Licht wird durch den Kegel begrenzt
      - Lichtintensität wird am Äußeren des Kegels niedriger
  - Komponenten der Reflektion **Phong Reflection**
    - Ambient light
      - Aufleuchtung durch Umgebung, keine Spotlights
      - $L_{amb} = k_{amb} * i_{amb}$
        - k_{amb} gibt an, wie ein Objekt indirekt aufgehellt wird (Farbe)
        - i_{amb} gibt an, wie viel indirektes Licht vorhanden ist (Intensität)
    - Diffuse reflection
      - Reflektion von matten Oberflächen
      - **Lambert's cosine law: Die Intensität der Reflektion ist abhängig vom Eintreffwinkel der Lichtquelle**
      - $L_{diff} = k_{diff} * i_{in} * \cos\theta = k_{diff} * i_{in} * \langle n|l\rangle^+$
        - θ = Eintreffwinkel des Lichts zur Normalen l = Lichtrichtung
        - wenn $\langle n|l\rangle$ negativ / kleiner 0 ist, wird Fragment nicht vom Licht bestrahlt
    - Specular reflection
      - Reflektion von glänzenden Oberflächen
      - Licht wird kegelförmig reflektiert, da sonst nichts sichtbar wäre (Lichtquelle ist nur ein Punktt)
      - Intensität wird kleiner, je weiter man sich von der Hauptrichtung der Reflektion entfernt
      - $L_{spec} = k_{spec} * i_{in} * \cos^{n_s}\phi$ $\cos\phi = \langle r|v\rangle$ $r = 2 * n * \langle n|l\rangle - l$
        - $\boldsymbol{n_s}$ **= Shinyness exponent** bestimmt, wie stark die Reflektion ist
          - hoch: glänzender, niedrig: eher diffus, $n_s \to \infty$: Reflektion kleiner, aber nicht heller
    - gesammelte Formel:
      - bei N Lichtquellen: $L = k_{amb} i_{amb} + \sum_{i=1}^{N} i_{in,i}(k_{diff} * \langle n|l_i\rangle^+ + k_{spec} * (\langle v|r_i\rangle^+)^{n_{s,i}})$

Shading

- Flat Shading
  - Jede Fläche hat eine konstante Normale
  - Die Beleuchtung ist abhängig von der Normalen
  - Eine Farbe / Fläche
- Gouraud Shading
  - An Punkten vom Polygon wird Beleuchtung berechnet
  - durch Interpolation der Punkte wird die komplette Fläche berechnet
- Phong Shading (!= Phong Lighting)
  - Vertex-Punkte und Normalen werden interpoliert
  - Für jedes Fragment wird die Beleuchtung einzeln berechnet

Texturen

- Texturatlas:
  - viele Texturen für zusammenhängende Flächen (z.B. Charakter) werden in einer Textur zusammengefasst
  - Mapping der Dreiecke des Modells auf den Texturatlas
  - erstellt durch Designer, **Texture Painting** auf 3D-Modell, danach kann Textur „aufgeklappt“ werden
  - Textur Wrapping:
    - Für Texturen, die außerhalb des definierten Bereichs sind, werden verschiedene Modes aktiviert
    - So wird die Textur z.B. wiederholt, gestreckt, gespiegelt etc.
- Texture Sampling:
  - Verfahren:
    - Simple Solution
      - Pixel nimmt die Farbe des Texels der am nähesten da ist (Mode: `GL_NEAREST`)
      - Problem: detaillierte Texturen werden stufig / pixelig
    - Bilinear interpolation
      - Durch Interpolation werden die 4 umliegenden Pixel einberechnet
  - Probleme:
    - Magnification Problem: niedrig aufgelöste Texttur, kleine Pixel -> scharfe Kanten gehen verloren
    - Minification-Problem: hoch aufgelöste Textur, weite Entfernung (Pixel größer als Texel)
      - Textur geht verloren
  - Mip Mapping:
    - Texturen werden im Voraus herunterskaliert → meist schrittweiße in 2 x 2 zu 1 x 1 Schritten
    - Je nach Pixelgröße wird entsprechendes Texturlevel zum Einfärben benutzt
    - Level Selection: $L = \log_2 p + L_{bias}$
      - p = ratio projected pixel to texel size
      - L_{bias} wird festgelegt, entweder mehr aliasing oder stärkere Verunschärfung
      - Problem:
        - Niedrig aufgelöste Texturen können verwischen
        - Dadurch entsteht ein „harter“ und erkennbarer Übergang
    - Level-Selection verfeinern durch Trilineare Interpolation zwischen $\lfloor L \rfloor$ und $\lceil L \rceil$
      - jeweilige lineare Interpolation der Farbe im oberen & unteren Level
      - lineare Interpolation zwischen den beiden Farbwerten
- GL Befehle bis hier:
  - Textur abrufen `vec4 texture(sampler, tex_coord)`
  - Texturvariable `uniform sampler2D diffuse_texture`
  - Texturkoordinaten `in vec2 tc`
  - Farbzuweißung `vec3 k_diff = texture(diffuse_texture, tc)`
    - Farbwerte speichern `texture(...).rgb`
  - Texture wrapping `glTexParameteri(target, pname, param)`
    - Modes: `GL_REPEAT, GL_MIRRORED_REPEAT, GL_GLAMP_TO_EDGE, …`
    - Sampling Verfahren: `GL_NEAREST & GL_LINEAR`

- Perspektive:
  - Interpolation im NDC != Interpolation im Eye Space
- Texturarten:
  - **Diffuse**: Details und Farben für die Geometrie
  - **Specular**: Glanz & Mattierung für die Textur
  - **Mask**: Gibt vor, welcher Teil der Textur benutzt wird

Verschiedene zusätzliche Texturen:

- Normal Maps zum Speichern der Normalen
  - Object Space
    - Normalen werden auf dem Objekt für jedes Fragment festgelegt
    - Berechnung: `vec3 n = vec3(2)*x – vec3(1)`
      - x sind „Farbwerte“, die aus der Textur gelesen wurden mittels `texture(...).rgb`
  - Tangent Space (viel besser)
    - Basis: TBN-Matrix (Achsenbezeichnung
    - Nicht nur Normale, sondern auch Tangentiale Achsen werden in der Textur abgelegt
    - Berechnung der T- und B-Achsen durch die Differenzen des Dreiecks in u- und v-Richtung
      - $\begin{pmatrix} e_{1,x} & e_{1,y} & e_{1,z} \\ e_{2,x} & e_{2,y} & e_{2,z} \end{pmatrix} = \begin{pmatrix} u_1 & v_1 \\ u_2 & v_2 \end{pmatrix} * \begin{pmatrix} T_x & T_y & T_z \\ B_x & B_y & B_z \end{pmatrix}$
        - e sind Strecken b-a bzw. c-a, u (T-Achse) und v (B-Achse) die zugehörigen Differenzen
    - Normale n wird aus Tangent-Space genommen, damit wird die Beleuchtung berechnet
  - Probleme:
    - Mip-Mapping funktioniert nicht gut
      - Normalen können sich zwischen 2 Texeln stark unterscheiden, Mittelwert hat andere Länge & Richtung
- Bump Maps
  - Höhe der einzelnen Texel wird gespeichert
  - „Normalhöhe“ ist bei ½
  - Wird meisten zu Normal Maps konvertiert
- Environment Maps
  - Skybox
    - Cube Maps
      - hat verschiedene Texturen für verschiedene primäre Richtungen
        - GL: `texture(samplerCube...)`
    - Longitude / Latitude Maps
      - für kompletten Himmel
      - von Richtung d aus kann durch folgende Formeln die Texturkoordinate berechnet werden:
        - $u = \frac{1}{2} + \frac{\tan^{-1}(d_z, d_x)}{2\pi}$
        - $v = \frac{1}{2} + \frac{\sin^{-1} d_y}{\pi}$
  - Billboards
    - Hintergrundobjekte, die nicht viel Beachtung bekommen
    - durch Mask-Textures kann einfach viel Hintergrund dargestellt werden
    - oft bei Gras der Fall

Shadow Mapping

- globaler Effekt → Objekte die Schatten werfen sind in der lokalen Beleuchtung nicht verfügbar
- Idee: Bild aus Sicht der Lichtquelle rendern
- Render-to-Texture:
  - Framebuffer wird erstellt `glGenFramebuffers(size, *buffer)`
  - Framebuffer wird gebunden `glBindFramebuffer(GL_FRAMEBUFFER, framebuffer)`
  - Texturen anhängen
    `glFramebufferTexture2D(target, attachment, textarget, texture, level)`
    - attachments:
      - color `GL_COLOR_ATTACHMENTO`
      - depht buffer texture `GL_DEPTH_ATTACHMENT`
    - textarget `GL_TEXTURE_2D`
  - im Framebuffer:
    - Viewport setzen `glViewport(x, y, width, height)`
    - Output bestimmen `glDrawBuffers(size, *bufs)`
- Rendern der Shadow Map
  - Zusätzlich zu shading point, camera point und light direction wird die kürzeste Distanz zum Licht gespeichert
    - zweifaches rendern zum Bestimmen der kürzesten Distanz
    - zum Rendern: `glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR)`
- Berechnung:
  - Berechnung der NDCs für Kamera und Licht
  - Vergleich der Z-Koordinaten von Punkt v_1 und Punkt v_2 in Licht-NDC
    - wenn $v_1 > v_2$ → **v_1 liegt im Schatten von v_2**
- Benutzung:
  - Texture Lookup: `d = texture(shadowmap, tc).r`
  - Shadow Lookup: `float s = texture(shadowmap, tc).r`
  - aktuell rasterisiertes Fragment wird in die Shadowmap projiziert und mit dem Wert verglichen
- Probleme:
  - Shadow Acne
    - durch grobe Rasterisierung werden Punkte teilweise als „im Schatten“ angesehen, obwohl sie das nicht sind
    - gelöst durch Verrechnen der Pixel mit Offset
      - Aktivieren: `glEnable(GL_POLYGON_OFFSET_FILL)`
      - Offset: `glPolygonOffset(factor, units)`
      - Deaktivieren: `glDisable(GL_POLYGON_OFFSET_FILL)`
  - Resolution Missmatch
    - Aliasing ist Problem
    - Auflösung ist teilweise zu schlecht auf kurze, aber zu hoch auf weite Entfernungen
    - Mip-Mapping nicht gut möglich, da nach ein paar Stufen nichts mehr von der Geometrie übrig bleibt
- Interpolation
  - für die GPU muss extra spezifiziert werden, dass der sampler eine shadowmap ist
    - GL: `texture(sampler2Dshadow sampler, P, [float bias])`
    - dadurch wird der shadow lookup abgefragt
  - Interpolation in der `texture(...)` Funktion funktioniert bei Tiefenwerten nicht, weil der Durchschnitt keine sinnvolle Funktion hat
  - wenn Hardwarefiltering angeschaltet ist, wird aus 4 verschiedenen Tiefenwertberechnungen der interpolierte Wert berechnet

Formeln:

- Gamma-Korrektur: $x_{out} = \sqrt[\gamma]{x_{in}}$
- lerp: $p_z = (1-t) * a_z + t * b_z = a_z + t * (b_z - a_z)$
  - $t_{AC}(P) = \frac{P_y - A_y}{C_y - A_y}$
- Baryzentrische Koordinaten: $p = a + \beta * (b-a) + \gamma * (c-a)$
- renormalisieren: $n' = (A^{-1})^T * n$
- near distance n, aspect ratio a, field of view θ
  - $w = a * n * \tan\frac{\theta}{2} \quad = r = l$
  - $h = n * \tan\frac{\theta}{2} \quad = b = t$
  - $a = \frac{w}{h}$
- z-Koordinate im NDC: $z' = -\frac{n+f}{n-f} - \frac{2*f*n}{z*(n-f)}$
- Phong Lighting:
  - $L_{amb} = k_{amb} * i_{amb}$
  - $L_{diff} = k_{diff} * i_{in} * \cos\theta = k_{diff} * i_{in} * \langle n|l\rangle^+$
  - $L_{spec} = k_{spec} * i_{in} * \cos^{n_s}\phi \qquad \cos\phi = \langle r|v\rangle \qquad r = 2 * n * \langle n|l\rangle - l$
  - bei N Lichtquellen: $L = k_{amb} i_{amb} + \sum_{i=1}^{N} i_{in,i}(k_{diff} * \langle n|l_i\rangle^+ + k_{spec} * (\langle v|r_i\rangle^+)^{n_{s,i}})$

$$\{\triangle_{local}\} \xrightarrow{\boldsymbol{M}} \{\triangle_{world}\} \xrightarrow{\boldsymbol{V}} \{\triangle_{eye}\} \xrightarrow{\boldsymbol{P}} \{\triangle_{clip}\} \xrightarrow{dehomo-genisieren} \{\triangle_{NDC}\} \xrightarrow{W} \{\triangle_{viewport}\} \xrightarrow{rasterization} [\![\square]\!]$$

OpenGL Shading Language (GLSL)

- die `main()`-Funktion wird jedes Mal, wenn der Shader aufgerufen wird, ausgeführt
- Versionsspezifizierung `#version 130`
- Spezifiziert über Main-Funktion
  - Input `in vec4 local_vertex`
  - Output `out vec4 out_color`
  - Daten, die unabhängig vom Punkt sind `uniform mat4 proj`
- main-Funktion `void main () {...}`
- implizite output-Variable für Punkt `gl_Position`

Beispiel Phong Shading mit point-light:

```
// Vertex Shader Phong Shading mit Phong Lighting
in vec3 vert_pos; // vertex position
in vec3 norm; // fragment normal

uniform mat4 P; // Projection Matrix
uniform mat4 V; // Viewing Matrix
uniform mat4 M; // Model Matrix
uniform mat4 M_normal; // Model Matrix for normal, inverted model matrix

out vec4 pos_ws; // World Space position
out vec3 norm_ws; // World Space normal

int main() {
    pos_ws = M * vec4(vert_pos, 1.0); // position in World Space

    norm_ws = normalize(M_normal * vec4(norm, 1.0)).xyz; // normal in World Space

    gl_Position = P * V * pos_ws; // point position in clip space
}
```

```
// Fragment Shader Phong Shading mit Phong Lighting
in vec3 pos_ws; // position in world space
in vec3 norm_ws; // fragment normal in world space

uniform vec3 pointlight_pos; // position of point light
uniform vec3 cam_pos; // position of cam
// color & intensity for diffuse lighting
uniform float k_diff;
uniform float i_diff;
// color & intensity for specluar lighting
uniform float k_spec;
uniform float i_spec;
uniform float n_s; // shininess exponent

out float color;

int main() {
    // diffuse lighting
    vec3 n = normalize(norm_ws); // normalize normal
    vec3 l = normalize(pointlight_pos - pos_ws); // normalize pointlight vector to point -
-> light vector
    float diff = k_diff * i_diff * max(dot(n, l), 0); // diffuse lighting

    // specular lighting
    vec3 v = normalize(cam_pos - pos_ws); //normalized view-vector
    vec3 r = 2 * n * dot(n, l) - l; // normalized r-vector (r = 2*n*<n|l> - l)
    float spec = k_spec * i_spec * pow(dot(r,v), n_s); // specular lighting

    color = diff + spec; // color = amb + diff + spec
}
```